計算機アーキテクチャ　104−10
システムソフトウェアと
オペレーティング・システム　62−10
（１９９４．　１．　２８）

BSD UNIXの下でのポータブルマルチスレッドライブラリPTLの実現

安倍 広多, 松浦 敏雄, 谷口 健一

大阪大学基礎工学部情報工学科

あらまし

筆者らは, BSD UNIXの下でマルチスレッド機構を実現するためのライブラリを作成した. ライブラリの設計に当たっては, コンテキストスイッチのオーバヘッドが小さい等のスレッドとしての基本的な性能を犠牲にすることなく, 様々なアーキテクチャ上に容易にインストールできることを目標とした. その実現においては, アーキテクチャに依存せずにスタックポインタを設定する方法や, スレッドスタックを自動的に拡張する方法等が問題となる. 本稿では, これらの問題に対する対処法について述べる.

作成したライブラリは, SunOS 4, Ultrix 4, DEC OSF/1, NEWS-OS 4, BSD/386 等, 数多くのアーキテクチャ上で動作することを確認している.

A Portable Library for POSIXthreads on BSD UNIX

Kota Abe, Toshio Matsuura, Kenichi Taniguchi

Dept. of Information and Computer Sciences,
Faculty of Engineering Science,
Osaka University

Machikaneyama 1-1, Toyonaka, Osaka 560 Japan.
E-mail: k-abe@ics.es.osaka-u.ac.jp

Abstract

We have implemented a portable library for POSIXthreads on BSD UNIX. In this paper, we have shown some implemetation problems and their solutions. A Goal of the library is to keep its portability with less overhead such as the time for context switching between threads. We have confirmed that it works on most BSD-based UNIX such as SunOS 4, Ultrix 4, DEC OSF/1, NEWS OS 4, BSD/386, etc.

1 まえがき

計算機ネットワークの普及に伴い, サーバクライアントモデルに基づくプログラムが広く用いられるようになってきた. このようなプログラムに限らず, 一般に複数の作業を協調しながら並行して動作させる必要があるプログラムは少なくない. UNIX の下でこのような処理を行なわせるには, それぞれの作業を別々のプロセスに割り当てて実行させるのが一般的であるが, 通常, UNIX のプロセスの生成や切替えには相当の時間を要し, このような作業を高速に実行させることは難しい. そこで, 一つの UNIX プロセスの中に, 概念的な, より小さなプロセス (**軽量プロセス**あるいは**スレッド**等と呼ばれる) を複数生成し, それぞれが独立した処理を担当できるような仕組みが利用されるようになってきた.

このような機構をカーネルレベルで実現しているOS もあるが [4][7], この方法ではシステムコールにかかるオーバヘッドのため, 速度の点からはユーザレベルで実現した方が効率が良いことが知られている [5]. ユーザレベルでの実現例として, SunOS の Light Weight Process (LWP) ライブラリ [9], フロリダ州立大 の SPARC 用ライブラリ [3] 等が存在するが, これらは, 特定のアーキテクチャに依存しており, 他のアーキテクチャでは動作しない. このため, 利用者は限られたアーキテクチャ上でしかマルチスレッドを利用したプログラムを動作させることが出来なかった.

そこで本研究では, 様々なアーキテクチャ上に容易に実装できるマルチスレッド機構を提供するライブラリの実現法を検討し, 作成した. このライブラリを PTL (Portable Thread Library) と呼ぶ. PTL は BSD スタイルのシグナル機構を備えた UNIX 上に実装可能であり, ほとんどの場合, make コマンドを実行するだけでインストールできる. ユーザはこれを利用することにより, スレッドを利用したプログラムを様々なアーキテクチャ上で動作させることが可能となる. 本稿では, PTL の実現上の問題点とその解決法について述べる.

2 PTL に対する要求とその実現

2.1 PTL に対する要求

PTL に対する要求項目を以下に示す.

- ライブラリ自身の移植性 — 特定の CPU に依存せず, なるべく多くのアーキテクチャで動作すること.
- 少ないオーバーヘッド — コンテキストスイッチやスレッドの生成等の頻繁に発生する処理が高速に実行されること. また, 入出力の実行時になるべくプロセス自体がブロックされないようにすること.
- インタフェースの互換性 — アプリケーションインタフェースが標準的であること.
- デバッギングの支援 — マルチスレッドを用いたプログラムは実行の流れが複雑なため, デバッグが一般に困難である. このため, デバッグを支援する機能があることが望ましい.

2.2 設計方針

上の要求を満たすため, 以下の方針に基づいて設計した.

- 移植性の確保 — 特定の CPU に依存するコーディングを避け, すべてを C 言語で記述する. また, アーキテクチャに依存する項目 (スタックの成長方向, 共有メモリの使用の可否, mprotect システムコールの使用の可否, シグナルハンドラを実行するのに必要なスタックのサイズ, _longjmp 関数が低位アドレスのスタックを持つ環境にジャンプできるか否か等) は, PTL のインストール時に自動的に調査し, 適切なシステムが構築できるようにする. また, 動作対象としては, もっとも普及していると考えられるバークレー版の UNIX[1] を対象とする.
- 高速化 — スピードを要求される個所では, システムコールの数を減らし, なるべく高速に動作するようにする. また, 入出力待ち中はブロックせずに時間を有効に利用できるようにする.
- 標準案への準拠 — PTL のアプリケーションインタフェースは, POSIX のスレッド拡張に対する規格案である P1003.4a/D6 (Pthread) [8] に準拠する.

[1] 正確には, BSD スタイルのシグナル処理が可能な UNIX

- 実行履歴の記録 — デバッグを支援するため, スレッドの実行履歴を保存し, 後で調査することが出来るようにする.

2.3 実現上の問題

スレッドは, アドレス空間, ファイル記述子等を共有しながら, それぞれ, CPU のレジスタ, 局所変数, スタックと, 独立した制御の流れを持つ. このような機構を UNIX の下で実現する際の主な問題点を以下に示す.

1. **スタックに関する問題**

   通常の UNIX プロセスでは, スタックがあふれた場合, カーネルが自動的にスタックセグメントを拡張する. しかし, スレッドのスタックはカーネルが管理していないため, スタックの割り当て, 拡張はライブラリ側で行なう必要がある. これらの操作をユーザレベルで実現することは容易ではない.

   また, スレッドを開始させるためには, スタックポインタをスレッドのスタックの底に設定する必要があるが, C 言語から CPU のスタックポインタを任意に変更することは一般に困難である. このため, この処理はアセンブラによって記述されることが多く, 移植性の障害となっている.

2. **シグナルに関する問題**

   シグナルが配送される際, カーネルは現在のコンテキストをスタックに保存するが, その際, スタックが不足していると, プロセスは異常終了してしまう. これは拡張可能なスタックを実現する際に障害となる.

3. **I/O に関する問題**

   UNIX では通常, 入出力 (I/O) の実行中はプロセス全体がブロックされ, I/O の完了で開放される. マルチスレッドのプログラムにおいては, あるスレッドの I/O 操作の実行によってプロセス全体がブロックしてしまうのは望ましくない.

3 スタック処理

スレッドのスタックの割り当て, 拡張の方法, 及び, スタックポインタの設定の方法を示す.

通常, スタックは高位アドレスから低位アドレスに向かって伸びて行く[2]. スタックの最上位アドレスを Stack bottom, 最下位アドレスを Stack limit と定義する.

3.1 スタックの割り当て

スレッドにはスレッドの作成時にプログラム中で指定したサイズのスタックを割り当てる. 一番簡単で, 移植性のあるスタックの割り当て方法は, ヒープ領域から割り当てることであるが, この方法では, スタックがあふれた際にプロセスが異常動作してしまう.

そこで, スタックのあふれ検出と, さらには自動拡張を行なうため, PTL ではヒープ領域から確保するスタック (ヒープメモリスタック) に加えて, 以下の 2 種類のスタックを提供している. これらのスタックでは, Stack limit より下位アドレスをアクセスするとシグナルが発生するため, スタックあふれの検出が可能である. ユーザはそれぞれのスタックの特徴とコストに応じて選択できる.

共有メモリスタック 共有メモリ機構をカーネルがサポートしている場合, 共有メモリセグメントをプロセスの仮想記憶空間に張り付け, その上にスタックを確保する. 共有メモリを張り付けるアドレスを比較的任意に選べるアーキテクチャの場合, スタックを自動的に拡張することも可能である (3.2 参照).

レッド・ゾーンスタック mprotect システムコールがサポートされている場合, スタックをヒープから確保し, Stack limit から下位アドレスの一定領域を読み書き禁止に設定することで, スタックのあふれを検出できる. この方法は文献 [3] 等でも使われている方法である.

3.2 スタックの自動拡張

一つのスタックは, 仮想アドレス空間上で連続していなければならない. また, 使用中のスタックにはスタックの仮想アドレスに依存した内容が含まれている可能性があるため, スタックを別の仮想アドレスに移すことは出来ない. このため, スレッドのスタッ

[2] PTL はスタックが低位アドレスから高位アドレスに向かって伸びるアーキテクチャにも対応しているが, 本稿中ではスタックは高位アドレスから低位アドレスに伸びるものとして述べる.

クがあふれた場合, あふれたスタックは移動出来ず, その場で拡張しなければならない.

ヒープメモリスタックや, レッド・ゾーンスタックでは, スタックの下位アドレス空間には別のデータが格納されているため, スタックを拡張することは不可能である. 一方, 共有メモリスタックの下位アドレス空間には通常, メモリが割り付けられていない. 共有メモリスタックがあふれた際に, この部分にメモリを張り付けることができ, かつスタックがあふれた際に実行しようとしていた命令を再実行することができれば, スタックのあふれから回復できる. 本ライブラリでは, 以下のようにして共有メモリスタックのあふれから回復を試みる.

1. PTL を初期化する際に, セグメンテーション違反を犯した際に発生するシグナル (SIGSEGV) を捕捉しておく[3].
2. スレッドが与えられた共有メモリスタックを使い果たして, メモリの割り付けられていない領域をアクセスすると, SIGSEGV シグナルが発生する.
3. SIGSEGV シグナルハンドラでは, より大きな共有メモリセグメントへスタックの内容を全てコピーし, もとの共有メモリセグメントと置き換え[4], 復帰する. これにより, スタックあふれを起こした命令から再開できる. (図 1)

スタックの拡張は, 次の条件を満たす UNIX で行なうことができる. (1) 共有メモリを張り付けるアドレスを比較的任意に[5]選べること. (2) シグナルハンドラから, シグナルが発生する直前のスタックポインタの値を取得できること (4.3参照). (3) SIGSEGV シグナルハンドラからセグメンテーション違反を起こしたアドレスを取得できること.

3.3 スタックポインタの設定方法

スレッドを開始する際には, スタックポインタを Stack bottom に設定しなければならない. 通常, このためにはアセンブラを用いて直接スタックポインタの値を設定するが, この方法では移植性が悪くなる. このため PTL では, この処理を BSD UNIX のシグナルスタック機能[6]を用いて, 以下のように行なう.

1. 確保したスレッドのスタック領域を sigstack システムコールを使ってシグナルスタックとして使用することを宣言する
2. 自分自身にシグナル (SIGUSR2) を送ってシグナルハンドラに制御を移す. この時, シグナルハンドラはシグナルスタック (すなわちスレッドのスタック領域) で実行されている.
3. シグナルハンドラからスレッドのスタート関数を呼び出す.

これによって, C 言語と UNIX システムコールだけでスタックポインタの設定が可能となった.

4 シグナル処理

Pthread では, 通常の UNIX の (プロセスレベル) シグナルは, 一定のルールで, スレッドに配送される (スレッドレベルシグナル) が, これを実現するため, PTL では以下のような処理を行う.

4.1 プロセスレベルシグナルの初期化

PTL の初期化時に, ほとんどのシグナルを, 内部に用意した汎用 (プロセスレベル) シグナルハンドラで捕捉するように準備しておく. また, 汎用シグナルハンドラ実行中に新たなシグナルが到着して別のハンドラが実行されるのを防ぐため, シグナルハンドラ実行時に全てのシグナルの配送をブロックするように手配しておく.

4.2 シグナルハンドラ用のスタックの確保

シグナルが到着した際, (プロセスレベルの) シグナルハンドラを実行するスタックの残りサイズが, カーネルのシグナル処理に必要な大きさ (アーキテクチャ依存) を下回っていると, プロセスは捕捉できない無効命令シグナル (SIGILL) を受け取って異常終了してしまう (共有メモリスタックやレッド・ゾーンスタック等, Stack limit より下位アドレスのメモリを

[3] SIGSEGV シグナルハンドラはシグナルスタックで実行されるように宣言しておく. ユーザスタックはあふれている可能性があるので, ユーザスタックの上でシグナルハンドラを動かすわけにはいかない. (4.2 参照)

[4] この際, Stack bottom のアドレスが変化しないように置き換える.

[5] ページサイズの整数倍アドレス等の緩やかな条件が望ましい.

[6] シグナルハンドラを実行するスタックとして, あらかじめ指定したメモリ領域を使用することが出来る機能

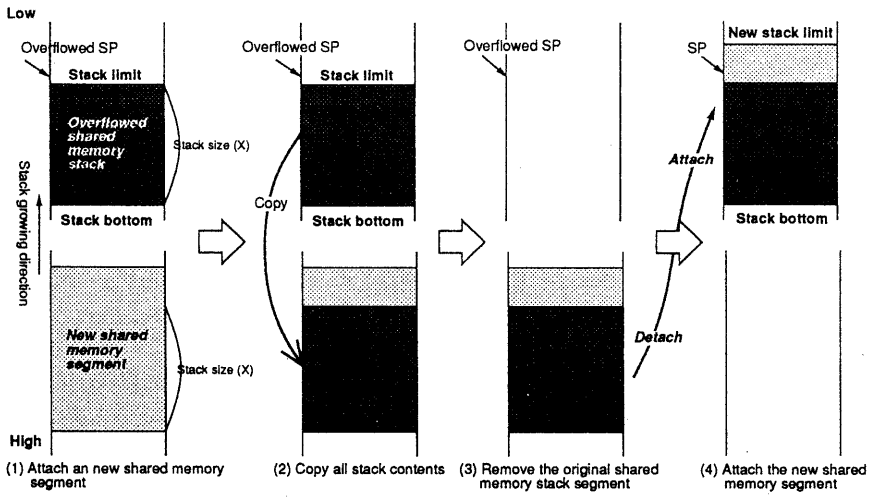

図 1: 共有メモリスタックの拡張

アクセスできない場合). あるいは, 他のメモリ領域を破壊する (ヒープ領域などからスタックを確保した場合). スレッドスタックを自動拡張するためには, この問題に対処しなければならない. これに対する解決法を以下に述べる.

シグナルの到着は予測できないため, 全てのシグナルのハンドラは, 充分な大きさを持つスタックで実行されることがあらかじめ保証されなくてはならない. このため, 全てのシグナルハンドラは, あふれる可能性のあるスレッドスタックではなく, 充分な大きさを持つシグナルスタックで実行されるように手配する.

シグナルの到着の際にカーネルによってシグナルスタックに積まれる情報は, 割り込まれたスレッドを再開するために必要であるが, シグナルによってコンテキストスイッチが引き起こされる場合, この情報は他のスレッドがスケジュールされた後に必要となる. このため, PTL では, シグナルスタックをスレッド毎に用意している. (これを**スレッドシグナルスタック** と呼ぶ). コンテキストスイッチの際には, プロセスレベルのシグナルハンドラがスレッドシグナルスタック上で実行されるようにシグナルスタックを切り替える.

スレッドスタックを拡張できないアーキテクチャでは, 汎用シグナルハンドラは, 通常のスレッドスタックの上で実行される[7] シグナルが到着した際に, カーネルによって積まれるコンテキスト情報は割り込まれたスレッドのスタックの上にあるため, シグナルによってコンテキストスイッチが起きる場合も問題なく後で割り込まれたスレッドのコンテキストを復元できる.

4.3 シグナルの配送

汎用シグナルハンドラでは, シグナルをどのように配送するかを決定する. シグナルの配送で問題となるのは, シグナル到着時に実行中だったスレッドにシグナルが配送され, ユーザのシグナルハンドラを実行しなければならない場合である.

スレッドスタックを拡張できるアーキテクチャの場合, 汎用シグナルハンドラは, スレッドシグナルスタック上で実行されているため (4.2参照), 以下のようにしてスタックをスレッドスタックに切り替えてからユーザのシグナルハンドラを実行する (図2 (1)).

1. 現在の, スレッドスタックの残りサイズをチェックし, カーネルがシグナルを処理するために必要なスタックサイズが存在することを確認する. もし, 不足すると判断した場合は, スレッドスタックを予め拡張しておく.

[7] この場合, シグナル到着時にスタックが不足して異常終了する可能性はある.

2. 汎用シグナルハンドラを実行する直前のスタックポインタ (図2 の Stack Pointer)[8] をスタックの底として, ライブラリ内部の関数 (_pthread_sigtramp) を呼び出す. これは, 3.3と同じようにシグナルスタックを変更して行なう.

3. スレッドスタックで実行されている _pthread_sigtramp では, スレッドのシグナルハンドラを実行している間に到着するプロセスレベルシグナルのため, シグナルスタック (の底) を, 図2の New signal stack bottom に設定しなおす. これによって, スレッドシグナルスタックの容量が減少することになるが, スレッドシグナルスタックはあらかじめ充分大きく確保してあるため, 4.2で述べたような事態が発生する危険は少ない. もし, 確実な安全性を求めるのならば, スレッドシグナルスタックの残りサイズをチェックし, 残り少なくなったときは新たなシグナルスタックのための領域を確保すれば良い.

   次に, プロセスレベルのシグナルマスク (この時点で, ほとんどのシグナルはブロックされている. 4.1参照) を新たなシグナルを受け取れるようにクリアしてから, スレッドのシグナルハンドラを呼び出す.

4. スレッドのシグナルハンドラが復帰したら, 図2のスレッドシグナルスタックの Used by UNIX Kernel 中に含まれる情報から, プロセスレベルのシグナルが発生した時点のコンテキストが復元され, スレッドの実行が再開される.

スレッドスタックを拡張できないアーキテクチャの場合は, 汎用シグナルハンドラはスレッドのスタックで実行されているため, 汎用シグナルハンドラは, 単純にスレッドのシグナルハンドラを呼び出すだけで良い (図2 (2)).

5 I/O 処理

ここでは, スレッドの I/O によってプロセス全体がブロックされないようにする方法 (ノンブロッキング I/O) について述べる.

UNIX では, I/O 関係のシステムコールが即座に終了しない場合, ブロックする代わりにエラーとしてリターンするように設定することが可能である (ノンブロッキング I/O モード). また, 後に I/O が可能となった時にシグナル (SIGIO) を発生させることもできる (ファイル I/O を除く).

PTL では, 使用する全てのファイル記述子をこのように設定[9]し, I/O が即座に終了しない場合は, I/O を実行したスレッドのみがブロックするように, I/O 関係のシステムコールを同名の関数で置き換えている.

また, 常に SIGIO シグナルを待ってブロックしている内部スレッドがあり, プロセスに SIGIO が配送されると I/O 可能なファイル記述子を調べ, I/O 待ちでブロックしているスレッドを開放する.

5.1 端末 I/O

端末 I/O では, 上で述べた問題に加え, 以下のような問題がある.

通常, バックグラウンドプロセス[10]による制御端末への入出力は, そのプロセスをサスペンドさせる[11]. しかし, マルチスレッドプログラムでは, 入出力を行なったスレッドのみがブロックされることが望ましい.

PTL では以下の処理を行なうことで, これを実現している.

- バックグラウンド入出力の際に発生するシグナル (SIGTTIN, SIGTTOU) を常にマスクしておく. これによって, バックグラウンド入出力を行なおうとした際, サスペンドされるかわりに, エラーとしてすぐにリターンするようになる.

- スレッドが端末 I/O を行なおうとしたときに, プロセスがバックグラウンドにいれば, そのスレッドをブロックさせる.

- シェルはプロセスをフォアグラウンドに回す際に SIGCONT シグナルを送る[12]. PTL では, このシグナルを受け取ると制御端末を調べ, プロセスがフォアグラウンドに回されていれば, ブロックしているスレッドを開放する.

[8] 通常, sigcontext 構造体の sc_sp メンバに格納されている

[9] PTL 内部で open システムコール等を覆い隠すルーチンを用意することによって行なっている

[10] ここでは, ジョブ制御によって裏に回されたプロセスのこと

[11] 設定によっては出力は行なわれ, サスペンドされない場合もある

[12] SIGCONT を送らないシェルも存在する. このようなシェルに対してはパッチを用意している.

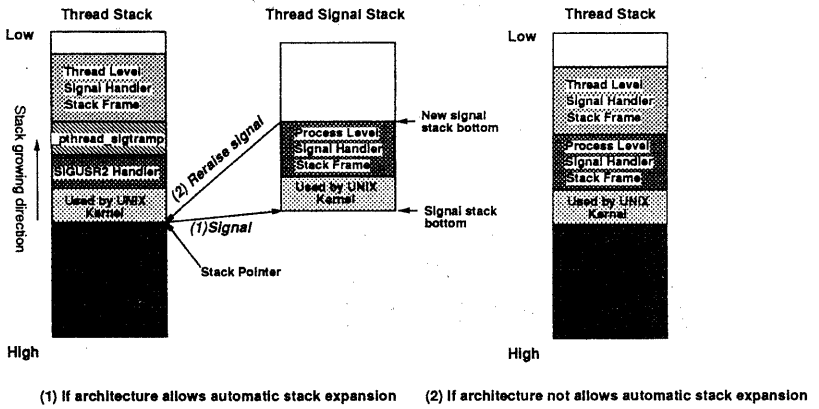

図 2: スレッドのシグナルハンドラの実行

これにより, バックグラウンドから端末 I/O を行なってもプロセス全体がブロックされることはない.

5.2 ファイル I/O

UNIX のプロセスは, ファイル I/O のシステムコール実行中は必ずブロックされる[13]. 要求した読み込みが, バッファキャッシュにヒットすれば[14], 直ちに終了するが, キャッシュミスの場合は比較的長時間ブロックされる (特に NFS の場合). キャッシュミスの際のブロックをユーザレベルで避けるためには, ファイル I/O を別プロセス経由で行ない, I/O 実行中はそのプロセスだけがブロックするようにする方法しかない.

しかし, ファイル I/O をすべて別プロセスに任せたのでは, バッファキャッシュにヒットした場合には, 通信のためのオーバヘッドのためにかえって効率が悪くなってしまう.

これに対し, バッファキャッシュにヒットする場合には I/O を直接行ない, ヒットしない場合のみ別プロセス経由で行なうという方法が考えられるが, 通常, UNIX ではキャッシュがヒットするかどうかをユーザは事前に知ることは出来ない.

そこで, PTL では, ライブラリレベルでファイルのキャッシングを行ない, 適当な大きさを持つブロックサイズで先読み, ライトバック制御を行なうことによって, 別プロセスの利用頻度を抑え, ファイル I/O を効果的に実行する機構を採用した (現在実装中である).

[13] SunOS 等, 非同期 I/O が可能な UNIX も存在する.

[14] 書き込みでは, バッファキャッシュへのコピーだけで終了する場合

6 実行履歴の表示

並列プログラムでは, プログラム中の複数の個所が同時に実行されるため, プロセス中で何が起きているのか把握しにくく, デバッグが困難である. 本ライブラリでは, デバッグを支援するため, スレッドの実行履歴を記憶しログファイルに出力する機能を提供している. 専用のプログラムがログを解析して視覚的に表示する (図 3).

これにより, 複数のスレッドの実行の流れを把握することが比較的容易となり, 実際にデバッグに役立っている.

7 あとがき

本稿では, 移植性の良いスレッドライブラリの実現上の問題と, その解決法について述べた. 作成したライブラリ (PTL) は, 現在以下のアーキテクチャで動作することを確認している.

SunOS 4, DEC Ultrix 4, DEC OSF/1 V1.3, SONY NEWS-OS 4, OMRON LUNA UniOS-B, OMRON LUNA-88K Mach 2.5, BSDI BSD/386, HP-UX

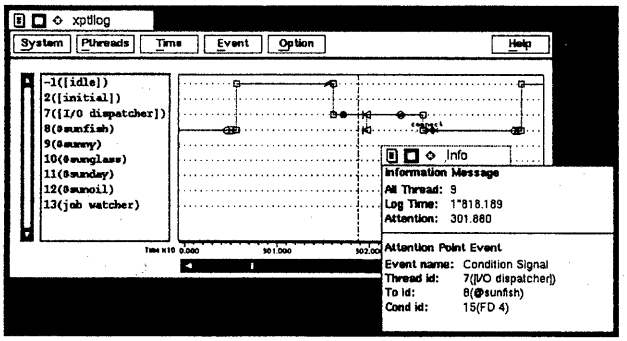

図 3: スレッドの実行履歴表示

現在 PTL の β 版を広域ネットワーク上で公開しており[15], 既に多くのユーザによって利用されている. また, 我々の研究室においても, PTL を利用したオブジェクトコードを生成する LOTOS [16] コンパイラを作成しており, 数千個のスレッドを並行に実行させるプログラムが稼働している.

PTL の性能については, 現在評価中である.

今後の課題としては, 動作中のプロセスのスレッドのモニタ機構の実現, 対話的なデバッグ環境の提供等が挙げられる.

謝辞 スレッドのログ表示プログラムの作成に協力して頂いた当研究室の城島貴弘君に感謝します.

参考文献

[1] 多田好克, 寺田実: "移植性·拡張性に優れた C のコルーチンライブラリー実現法", 電子情報通信学会論文誌 Vol.J-73-D-I No.12 pp.961–970 (1990-12).

[2] Frank Mueller: "Implementing POSIX Threads under UNIX", Proceedings of Second Software Engeneering Research Forum, pp.253–261 (1992).

[3] Frank Mueller: "A Library Implementation of POSIX Threads under UNIX", 1993 Winter USENIX.

[4] Eric C. Cooper and Richard P. Draves: "C Threads", Department of Computer Science Carnegie Mellon University, 1987.

[5] 新城靖, 清水康: "並列プログラムを対象とした軽量プロセスの実現方式", 情報処理学会論文誌 Vol.33 No.1, pp.64–73 (1992-1).

[6] Digital Equipment Corporation: "DECthreads – Guide to DECthreads", (1991).

[7] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier: "Multi-threaded Processes in CHORUS/MIX", Proceedings of EEUG Spring'90 Conference, pp.1–13.

[8] IEEE. Threads Extension for Portable Operating Systems (Draft 6), February 1992. P1003.4a/D6.

[9] Sun Microsystems: "System Services Overview", pp.71–106 (1988).

[10] D. Stein and D. Shah. "Implementing Lightweight Threads", Proceedings of the USENIX Conference, pages 1–10 (1992).

[11] 安倍広多, 松浦敏雄: "移植性に優れた軽量プロセスライブラリの実装法", Proceedings of the 21st jus UNIX Symposium, pp.78–89 (1993-7).

[15] `ftp.center.osaka-u.ac.jp:/PTL/`
[16] 分散システムや通信システムの仕様記述言語